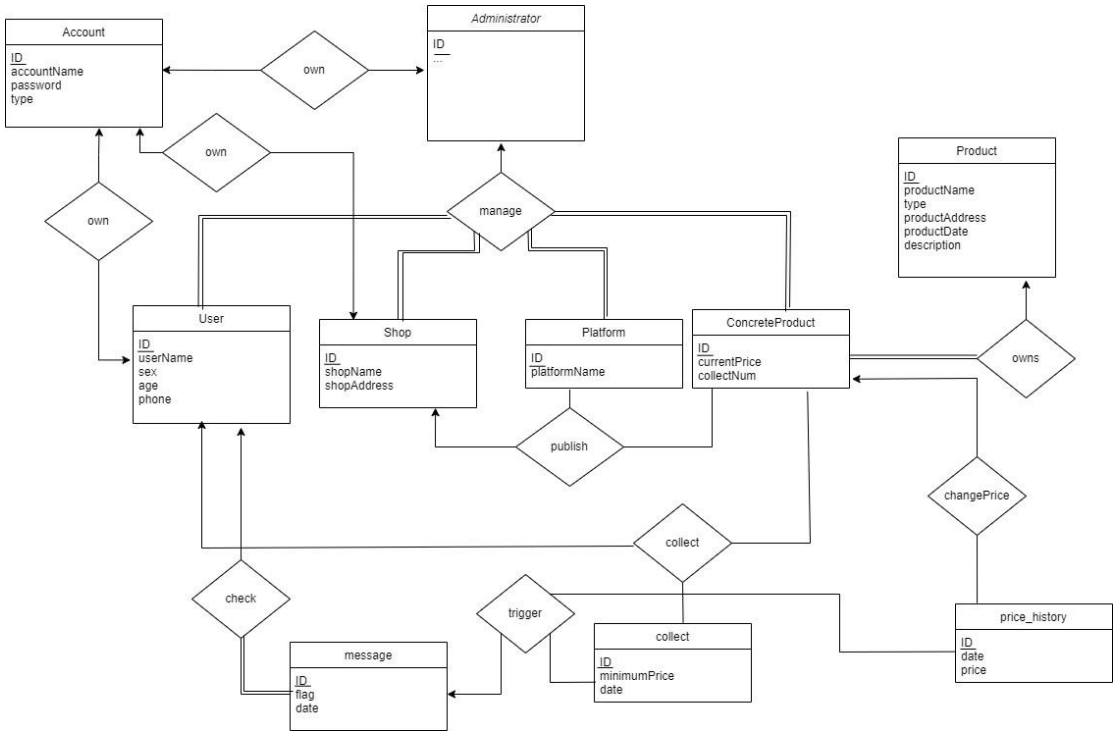
数据库说明文档

一．数据库 ER 图说明

ConcreteProduct 是具体商品，是 Product 商品的具体实现，也就是说 Product 代表一类商品（productName，type 等属性都相同），而 ConcreteProduct 是不同商家不同平台，针对该类商品设定不同价格下产生的单个具体商品，这样是为了减少数据冗余，否则记录中会有很多相同的如 productName，type 等等这些商品基本属性。Account 用于登录，User 和 Shop 和 Administrator 都有自己的信息，但通过绑定 Account 来进行登录。

Administrator 管理员可以对 User、Platform、Shop、ConcreteProduct 进行管理(manage),User，Shop，Administrator 通过绑定(own)Account 来登录。User 用户可以收藏(collect)具体商品，生成收藏记录。商品是由 Shop 商家选择平台来进行发布(publish)，生成具体商品与商品类（若需要），并且商家能够修改商品价格(changePrice)，会生成 price_history 历史价格表，用户在查询时也可以获得每个商品的历史价格表。当 ConcreteProduct 的 currentPrice 低于 collect 中的最低预期价格 minimumPrice 时，则通过触发器(trigger)生成 Message，用户可查看(check)降价提醒消息。



二．数据库建表

以下为建表语句，其中指定了各表各字段的名称与具体类型，字段名称为了与 Springboot 操作相对应，只能设置为小写以及下划线命名的形式。此外，对于每个表的 id 设置主键以及自增，并将某些字段正确地设置外键以实现表间关联与级联删除，如 concrete_product 表的 shopid 设置外键关联 shop 表，在删除 shop 表的某 id 时与之关联的 concrete_product 表中的记录也会删除。此外，还在相关表设置了索引以及实现消息触发器（具体说明在三里）

```
-- 创建 account 表
CREATE TABLE account (  id INT PRIMARY
KEY AUTO_INCREMENT,  account_name
VARCHAR(50),  password VARCHAR(50),
type INT,  infoid INT,
   INDEX idx_account_id (id)
);

-- 创建 user 表
CREATE TABLE user (  id INT PRIMARY KEY
AUTO_INCREMENT,  user_name
VARCHAR(50),  age INT,  sex
VARCHAR(10),  phone VARCHAR(20),
   INDEX idx_user_id (id)
);

-- 创建 shop 表
CREATE TABLE shop (  id INT PRIMARY
KEY AUTO_INCREMENT,  shop_name
VARCHAR(50),  shop_address
VARCHAR(50),
  INDEX idx_shop_id (id)

);
```

```
-- 创建 platform 表
CREATE TABLE platform (  id INT PRIMARY
KEY AUTO_INCREMENT,  platform_name
VARCHAR(50),  INDEX idx_platform_id (id)
);

-- 创建 product 表
CREATE TABLE product (  id INT PRIMARY
KEY AUTO_INCREMENT,  product_name
VARCHAR(50),  type VARCHAR(50),
product_address VARCHAR(50),
product_date VARCHAR(50),  description
VARCHAR(50),
  INDEX idx_product_id (id)
);

-- 创建 concrete_product 表
CREATE TABLE concrete_product (  id INT
PRIMARY KEY AUTO_INCREMENT,  shopid
INT,  platformid INT,  productid INT,
current_price FLOAT,  collect_num INT,
  INDEX idx_concrete_product_id (id),
  INDEX idx_shopid (shopid),
  INDEX idx_platformid (platformid),
  INDEX idx_productid (productid),
  FOREIGN KEY (shopid) REFERENCES shop(id) ON DELETE CASCADE,
  FOREIGN KEY (platformid) REFERENCES platform(id) ON DELETE CASCADE,
  FOREIGN KEY (productid) REFERENCES product(id) ON DELETE CASCADE
);

-- 创建 collect 表
CREATE TABLE collect (  id INT PRIMARY
KEY AUTO_INCREMENT,  userid INT,
```

```
concrete_productid INT,  minimum_price
FLOAT,  date DATE,
 INDEX idx_collect_id (id),
 INDEX idx_userid (userid),
 INDEX idx_concrete_productid (concrete_productid),
 FOREIGN KEY (userid) REFERENCES user(id) ON DELETE CASCADE,
 FOREIGN KEY (concrete_productid) REFERENCES concrete_product(id) ON DELETE
CASCADE
);

-- 创建 message 表
CREATE TABLE message (  id INT PRIMARY
KEY AUTO_INCREMENT,
concrete_productid INT,  date DATE,  flag
INT,  userid INT,
 INDEX idx_message_id (id),
 INDEX idx_concrete_productid (concrete_productid),
 INDEX idx_userid (userid),
 FOREIGN KEY (concrete_productid) REFERENCES concrete_product(id) ON DELETE
CASCADE,
 FOREIGN KEY (userid) REFERENCES user(id) ON DELETE CASCADE
);

-- 创建 price_history 表
CREATE TABLE price_history (  id INT
PRIMARY KEY AUTO_INCREMENT,
concrete_productid INT,  price FLOAT,
date DATE,
 INDEX idx_price_history_id (id),
 INDEX idx_concrete_productid (concrete_productid),
 FOREIGN KEY (concrete_productid) REFERENCES concrete_product(id) ON DELETE
CASCADE
);

-- 创建触发器
DELIMITER //
```

```
CREATE TRIGGER check_trigger_message
AFTER INSERT ON price_history
FOR EACH ROW
BEGIN
/*创建一个变量，用来保存当前行中 minimum_price 的值*/
  DECLARE tempprice FLOAT;
      /*创建一个变量，用来保存当前行中 userid 的值*/
  DECLARE tempid INT;
  /*创建游标结束标志变量*/
   DECLARE v_done int DEFAULT FALSE;
  -- 获取 collect 表中符合条件的所有行，并循环处理
  DECLARE cur CURSOR FOR
    SELECT minimum_price, userid FROM collect WHERE concrete_productid =
NEW.concrete_productid;
            /*设置游标结束时 v_done 的值为 true，可以 v_done 来判断游标是否结束了*/
  DECLARE CONTINUE HANDLER FOR NOT FOUND SET v_done=TRUE;

  OPEN cur;
       a:LOOP
            FETCH cur INTO tempprice,tempid;
       if v_done THEN           LEAVE a;
            END IF;

            IF tempprice > NEW.price THEN
      INSERT INTO message (concrete_productid, date, flag, userid)
      VALUES (NEW.concrete_productid, NEW.date, 0, tempid);
    END IF;
       END LOOP;
       CLOSE cur;
END //
DELIMITER ;
```

三．索引说明、触发器说明、核心功能 SQL 语句说明、事务说明索引

说明:

索引定义在建表时进行，建表语句参考上面的脚本，主要就是在每个表的主键与外键设置了索引，索引类型为 NORMAL，索引方法为 BTREE。索引能够有效提升查询速度，在后面大规模数据测试比较中会具体体现。

触发器说明:

上面的建表语句中还实现了一个简单的触发器，用于生成对于收藏商品的降价提醒，主要逻辑就是修改价格后会在 price_history 表中插入新记录，每次插入新记录先通过 SELECT 语句获得 collect 表中收藏该商品的记录集合。用游标来遍历这个集合，判断插入的这条记录的 price 是否小于对应收藏的 minimum_price，若小于则生成对应的一条 message 存入 message 表中即可。

核心功能 Sql 语句说明:

收藏表：

```
@Query("SELECT new com.example.backend0.dto.FavoriteDTO(cp.ID,p .productName,s.shopName,pf .platformName,cp.currentPrice,c .minimumPrice) "+
        "FROM Collect c "+
        "JOIN ConcreteProduct  cp on cp .ID=c.concreteProductID "+
        "JOIN Shop s on s.ID=cp .shopID "+
        "JOIN Platform pf on pf .ID=cp .platformID " +
        "join Product p on p.ID=cp .productID "
        "where c.userID=:userID ")
List<FavoriteDTO> getFavoritesByUserID(Integer userID);
```

通过连接和 where 的条件语句来找到对应 userID 的收藏表的表项以及该收藏商品相关的 ID 主键，商品名，平台名，当前价格，最低期望价格。

```
@Query("SELECT new com.example.backend0.dto.CollectDTO(ph.concreteProductID, p.productName, s.shopName,pf.platformName,p.type,cp.currentPrice) " +
        "FROM Collect ph " +
        "JOIN ConcreteProduct cp on cp.ID=ph.concreteProductID "+
        "JOIN Product p ON cp.productID = p.ID " +
        "JOIN Shop s ON cp.shopID = s.ID " +
        "JOIN Platform pf ON cp.platformID = pf.ID " +
        "WHERE  ph.date between :startDate and :endDate")
```

List<CollectDTO> findCollectByWeek2(Date startDate,Date endDate);

通过条件语句“ph.date between :startDate and :endDate”以及连接，找到日期位于 startDate 和 endDate 之间的收藏信息，以满足不同时间跨度的查询要求。

具体商品表：

```
@Query("SELECT new com.example.backend0.dto.FullProductInfoDTO(cp.ID,p.productName,s.shopName,s.shopAddress,pf.platformName,p.type,cp.currentPrice,p.productAddress,p.productDate,p.description) " +
        "FROM ConcreteProduct cp " +
        "JOIN Product p ON cp.productID = p.ID " +
        "JOIN Shop s ON cp.shopID = s.ID " +
        "JOIN Platform pf ON cp.platformID = pf.ID " +
        "WHERE cp.ID = :concreteProductID")
FullProductInfoDTO findFullProductInfoByConcreteID(Integer concreteProductID);
```

通过连接找到 concretproductID 主键对应的商品的详细信息。

```
@Query("SELECT new com.example.backend0.dto.ShopProductDTO(cp.ID,p.productName,p.type,p.productAddress,p.productDate,p.description,cp.shopID,pf.platformName,cp.currentPrice) "+
        "from ConcreteProduct cp "+
        "JOIN Product p on cp.productID=p.ID "+
        "JOIN Platform pf on pf.ID=cp.platformID "+
        "where cp.shopID=:shopID")
List<ShopProductDTO> getConcreteProductsByShopID(Integer shopID);
```

通过连接以及条件语句，查询某个桑点下的所有商品的信息。

信息表：

```
@Query("SELECT new com.example.backend0.dto.MessageDTO(m.ID,p .productName,s .shopName,pf .platformName,cp .currentPrice,c.minimumPrice,m .date) "+
        "from Message m " +
        "join ConcreteProduct cp on cp.ID=m.concreteProductID " +
        "join Product p on cp.productID=p .ID " +
        "join Collect c on c .concreteProductID=cp .ID and c.userID=:userID "+
        "join  Platform pf on pf .ID=cp .platformID "+
        "join Shop s on s.ID=cp .shopID "+
        "where m.userID=:userID  ")
List<MessageDTO> getMessagesByUserID(Integer userID);
```

找到某个用户的所有信息以及其中包含的商品的对应信息。

价格历史表：

```
@Query("SELECT new com.example.backend0.dto.PriceHistoryDTO(ph.ID, ph.price, ph.date) " +
        "FROM PriceHistory ph " +
        "WHERE ph.concreteProductID = :concreteProductID " +
        "AND ph.date between :startDate and :endDate")
List<PriceHistoryDTO> findPriceHistoryByRange (Integer concreteProductID, Date startDate,Date endDate);
```

通过多个条件语句查找一定时间范围内的所有历史价格信息，以支持按周，月，年查询。

```
@Query("SELECT ph.price " +
    "FROM PriceHistory ph " +
    "WHERE ph.concreteProductID = :concreteProductId " +
    "AND ph.date <= :date " +
    "ORDER BY ph.date DESC " +  // 通过日期降序排列
    "LIMIT 1")  // 限制结果为一条记录
```

Float getPriceByDateAndConcreteProductId(Date date,Integer concreteProductId); 查找到某一天的商品价格，如果当天没有记录则向前查找到最近的时间的价格。

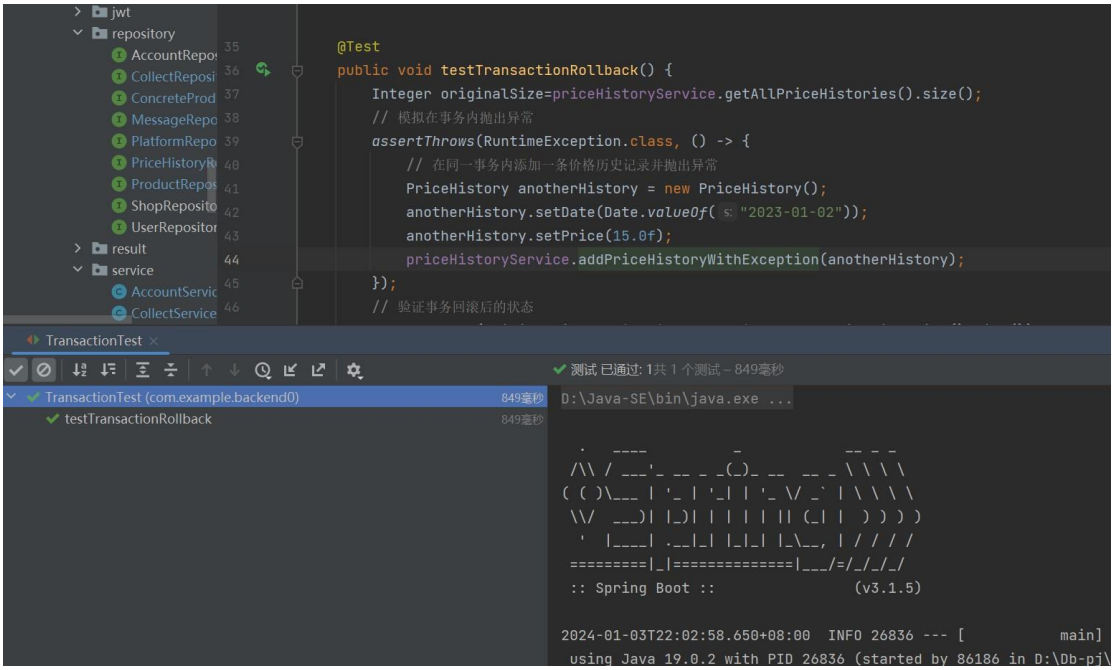
事务说明:

```
@Service
public class CollectService {
    2 个用法
    @Autowired
    CollectRepository collectRepository;
    xs-keju *
    @Transactional
    public Collect save(Collect collect) { return collectRepository.save(collect); }
    1 个用法  xs-keju *
    @Transactional
    public List<FavoriteDTO> getFavoritesByUserID(Integer userID){
        return collectRepository.getFavoritesByUserID(userID);
    }
}
```

```
@Transactional
@Query("SELECT new com.example.backend0.dto.FavoriteDTO(cp.ID,p .productName,s.shopNa
        "FROM Collect c " +
        "JOIN ConcreteProduct  cp on cp .ID=c.concreteProductID " +
        "JOIN Shop s on s.ID=cp .shopID " +
        "JOIN Platform pf on pf .ID=cp .platformID " +
        "join Product p on p.ID=cp .productID " +
        "where c.userID=:userID ")
List<FavoriteDTO> getFavoritesByUserID(Integer userID);
```

在 sql 查询语句和 Service 层的业务函数中都通过@Transactional 注解来实现事务的监测和回滚。

并且在单元测试中，针对 PriceHistoryService 进行了事务回滚的测试并且通过:



四．运行方法

前端 vue 项目运行方式:

1. 确保安装 node.js，可以从 Node.js 的官方网站（https://nodejs.org）下载并安装适用于您操作系统的版本。请保证 npm 版本在 10.2.3 以上，node 版本在 V20.10.0 以上。

2. 进入前端文件夹，终端下执行 npm install 后根据项目的`package.json`文件安装所需的依赖项。然后执行 npm run dev，正常情况下应该能直接运行，如下图显示。在网页中输入 http://127.0.0.1:70/即可访问网页，进行注册登录等系列操作（请确保后端开启）。

附前端使用注意事项：设置了 token 令牌，有效期为 3h，登录后很多操作都是使用 token 来进行身份验证和获取相关信息，故要保证 localstorage 可用。此外，未登录情况下并不能访问各角色主页，只能访问登录、注册、导入信息三个页面。

```
PS D:\desktop\databasePJ\databasepj> npm run dev

> databasepj@0.0.0 dev
> vite


  VITE v5.0.10  ready in 1208 ms

  ➜  Local:   http://127.0.0.1:70/
  ➜  press h + enter to show help
```

后端运行方式:

1. 确保安装了 java，可从官网下载并安装合适的 java 版本

https://www.oracle.com/java/technologies/downloads/，本项目使用的 java17。

2. 确保安装了 Mysql，可从官网下载并安装对应的 Mysql 数据库 https://dev.mysql.com/downloads/。
3. 修改后端项目文件夹中的 mysql 数据库配置，或者新建一个相同配置的数据库,并在数据库中执行前面提供的建表语句。

```
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/databasepj?characterEncoding=utf-8
spring.datasource.username=root
spring.datasource.password=123123
spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto=create
```

4. 如果修改了后端项目夹的数据库配置，则需要重新打包为 jar 包，jar 包名 backend.jar。
5. 如果新建了相同配置的数据库，则通过命令：java -jar bakend.jar 即可运行。

五．测试数据导入以及查询功能覆盖情况

测试数据导入:

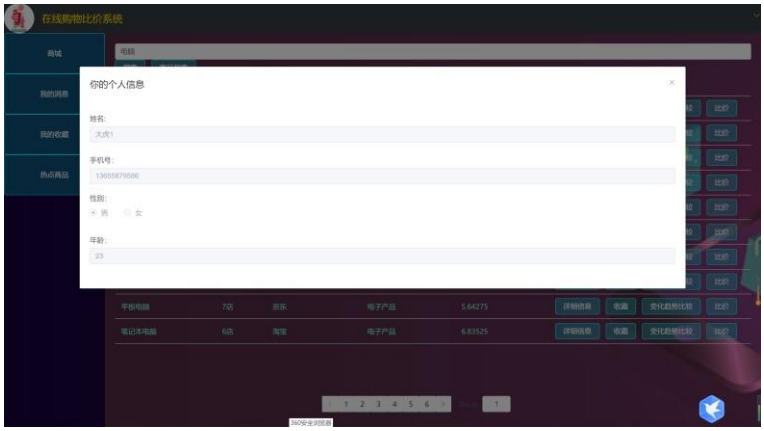
前端可以直接在网页里面输入 http://127.0.0.1:70/testaddinfo（会发现只有一个图标）就直接能够导入数据（请勿重复访问），或者 postman 对 http://localhost:8080/test/importFull 发送 POST 请求即可。该过程约 5 分钟，请耐心等待。如果是用 IDEA 打开后端的话，能够通过实时信息查看是否还在导入。

导入提供的管理员账号为 admin，密码 123123；导入提供的用户账号为 user0--user99，密码 111111；导入提供的商店账号为 shop0--shop29，密码 111111；也可以自己注册来进行测试。

系统工作流程（功能点）

1. 用户信息查询:

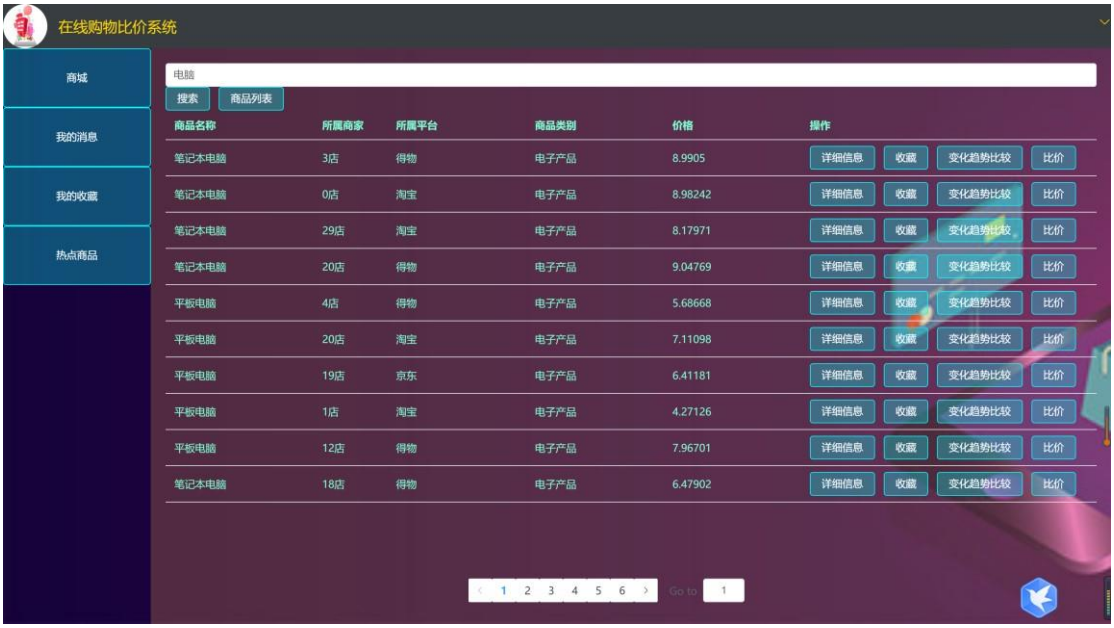
a. 用户可以查看自己账户的相关信息。



2. 商品搜索与浏览:

a. 用户可以使用搜索功能查找感兴趣的商品。

b. 搜索结果将显示一系列商品的简略信息，包括商品名称、商品所属商家、平台、价格等。



c. 用户可以查看商品更多详细信息。

3. 商品详细信息查看:

a. 用户可以查询一个商品的所有信息，包括商品描述、商家信息、当前价格等。

b. 用户查询商品的历史价格变化。

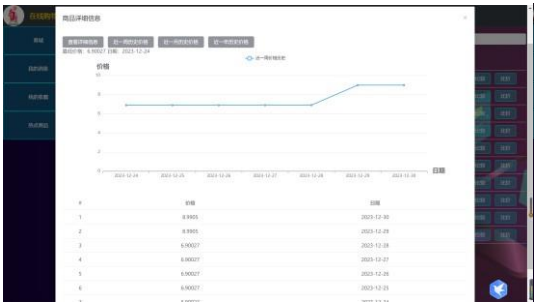
c. 注意，一个商品可以支持多个商家售卖，不同商家的价格可以不同。

d. 同样需要注意的，商家也可以将同一商品发布到不同的平台上售卖，且价格也可以有所不同

4. 商品价格变化查询:

a. 用户可以查询商品（选定平台/商家）一段时间内的价格变化，并突出展示最低价格及相应时间。

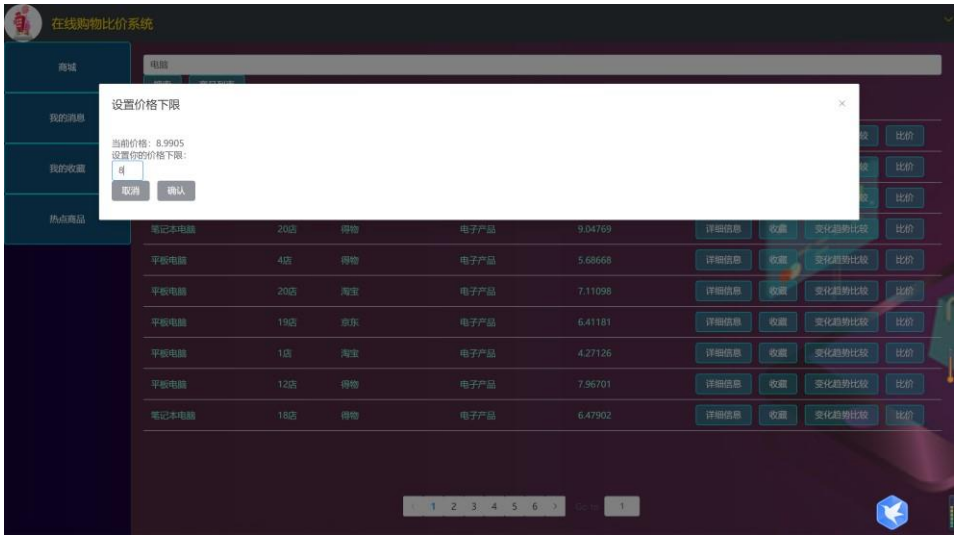
b. 用户可以选择不同时间跨度，如近一周、近一月、近一年，来查看价格趋势。



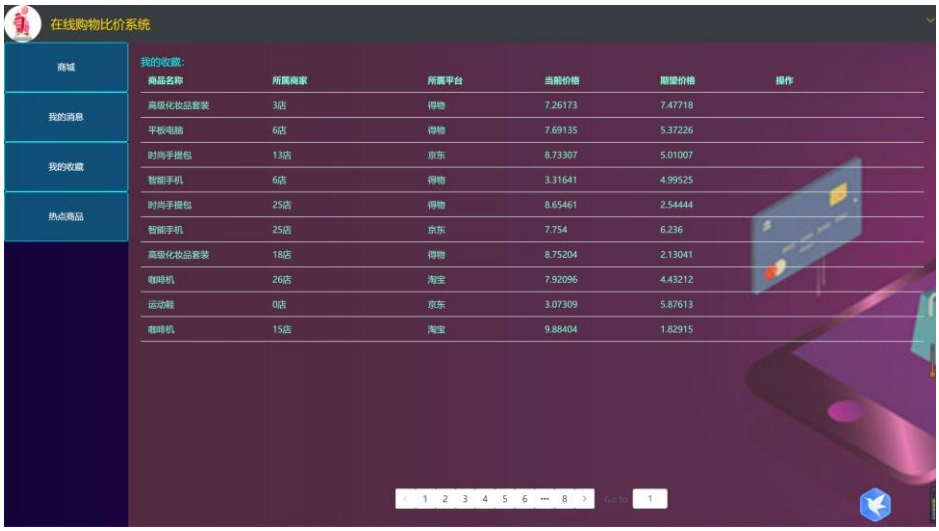
5. 收藏商品:

a. 用户可以收藏自己喜欢的商品（可以指定商家/平台）。

b. 用户可以为自己收藏的商品设定价格下限。

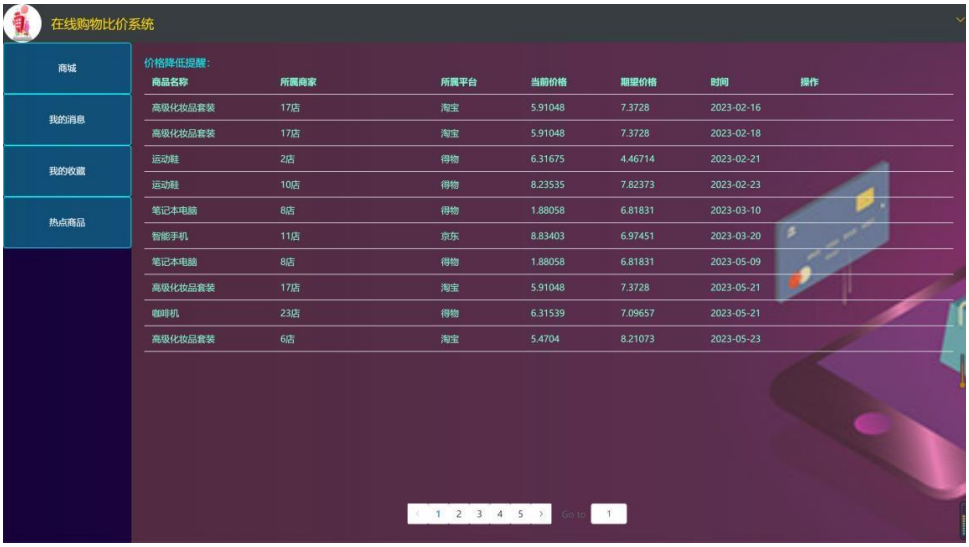


c. 当商品价格降低到等于或低于用户设定的价格下限时，系统会发送消息提醒用户。



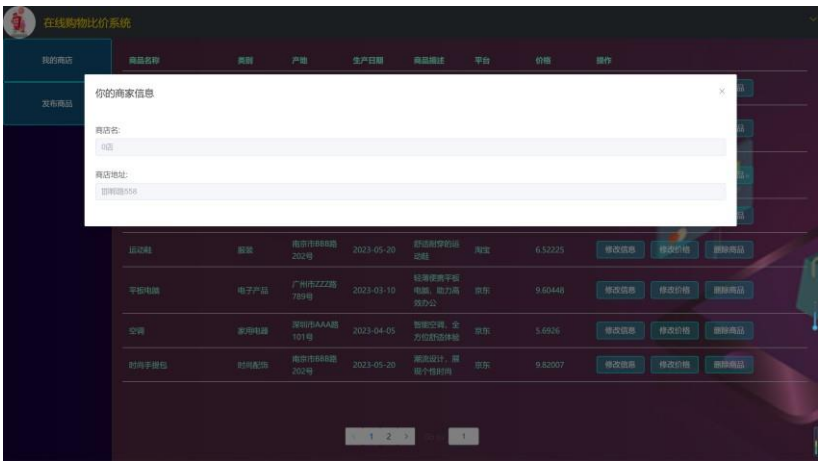
6. 查看消息列表：

a. 用户可以查看系统发送的消息列表，包括价格降低提醒（提醒中包含商品、售卖商家、平台、当前价格等必要信息即可）等。

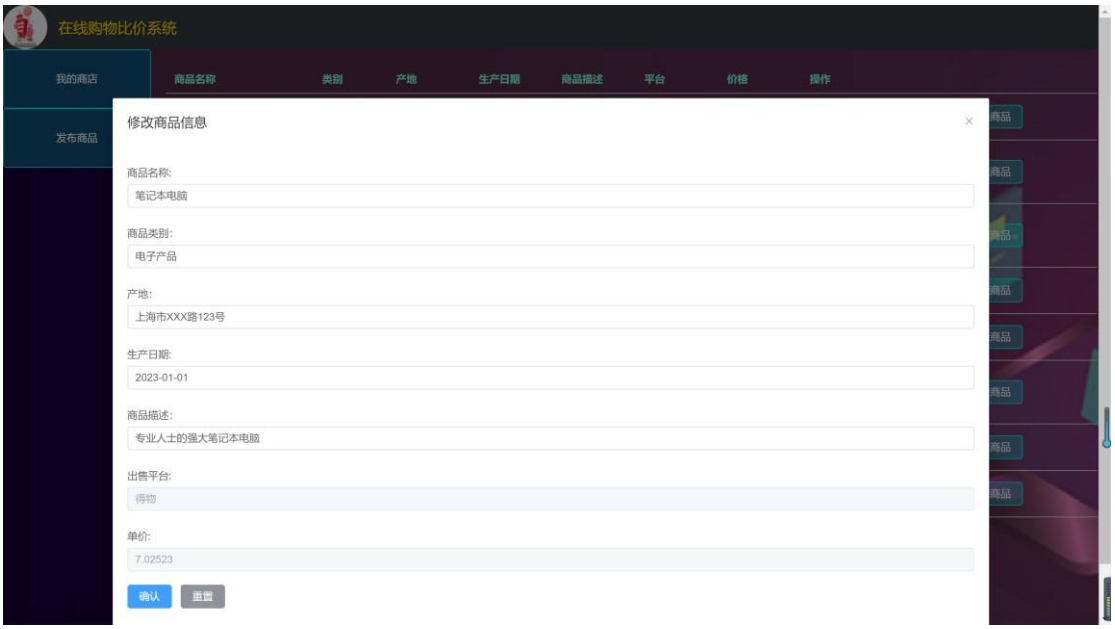


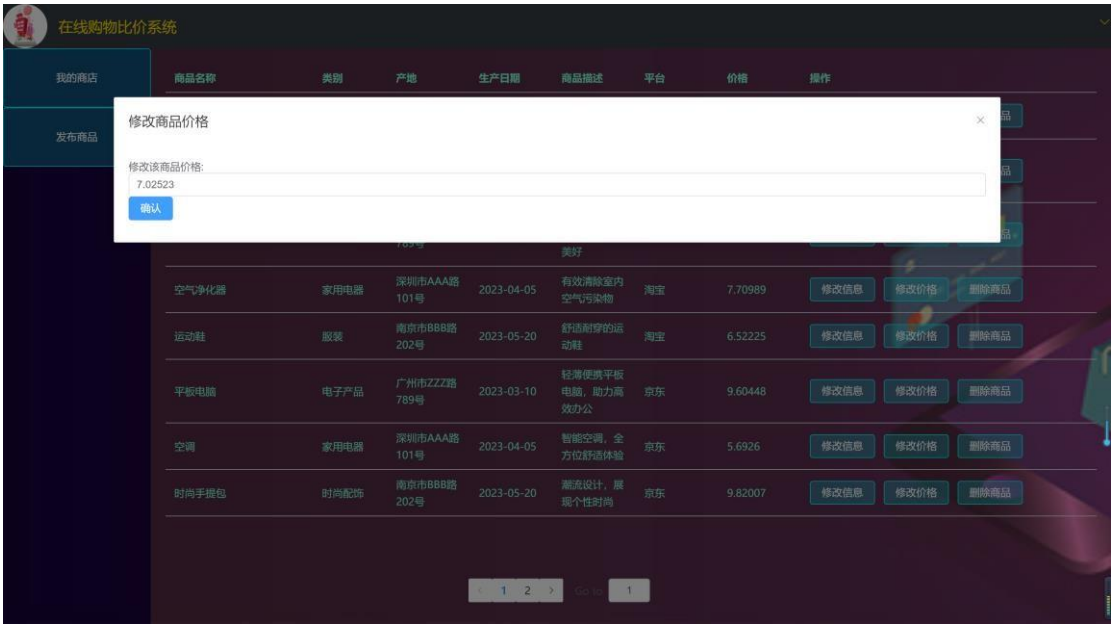
7. 商家信息管理与查询:

a. 商家可以查看自己的商家信息。



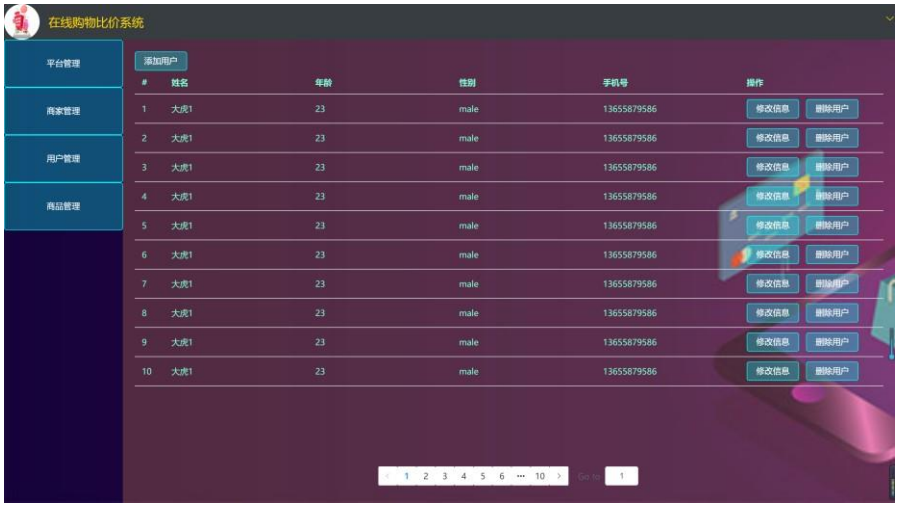
b. 商家可以发布、管理自己商品的信息和价格（在不同平台可以以不同的价格发布。对一个商品，商家每天在每个平台最多修改一次价格）。





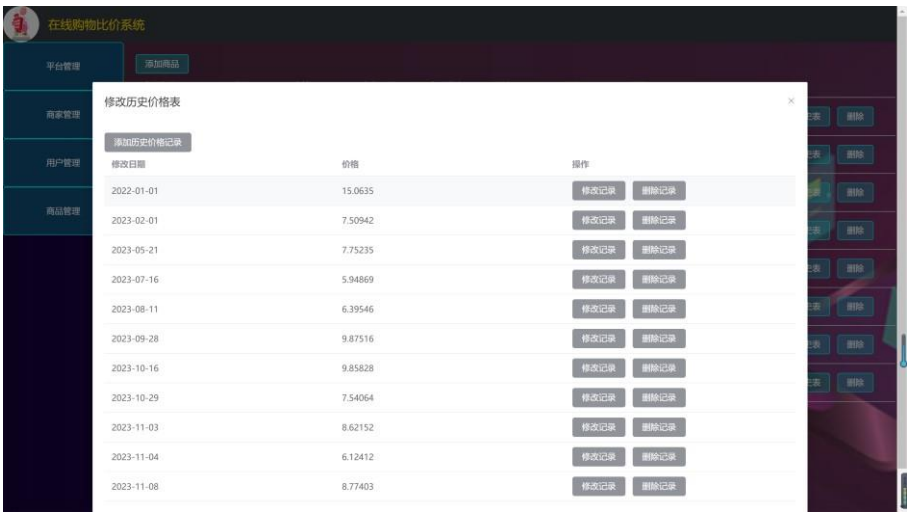
8. 管理员操作:

a. 管理员可以管理用户、商家的相关信息，如添加/删除/修改用户/商家。



b. 管理员可以管理平台的相关信息。

c. 管理员可以管理商品信息，包括价格历史



1. 基本查询需求

1) 查询某类商品的简略信息列表

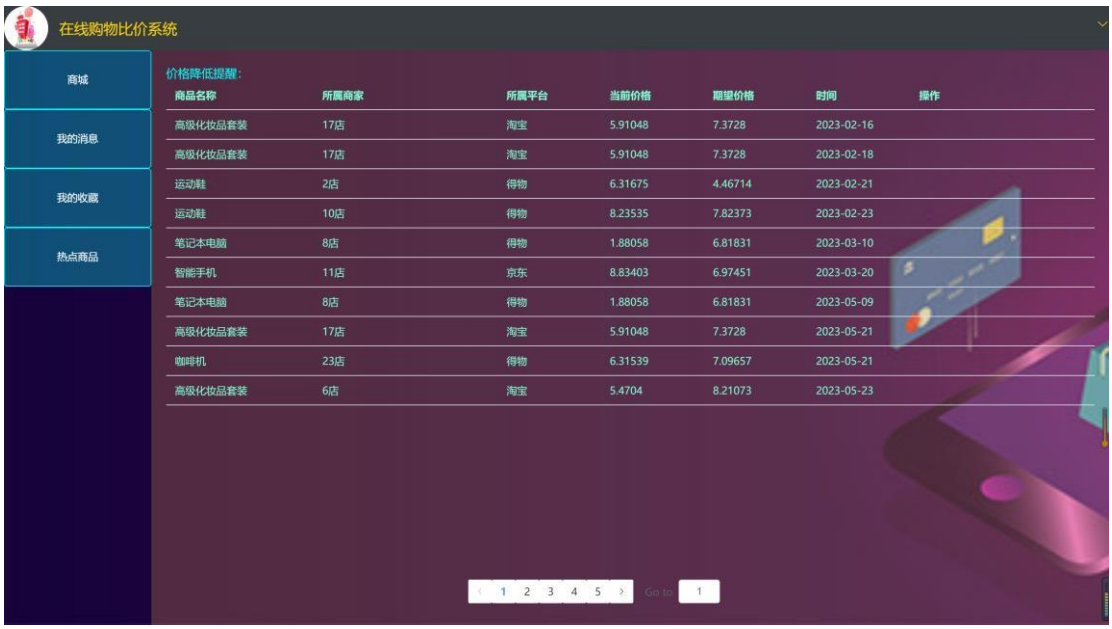
2) 查询某商品的详细信息

3) 查询某商品一段时间内的价格变化，支持按不同时间跨度（近一周，近一月，近一 年）进行筛选。界面上 不要求用可视化图表来显示价格变化趋势。但需要强化相关查 询能力，结果要能便于直接显示到界面（但不需要去实现这个界面；不需要针对特定 的 UI 组件去实现数据格式）。

如下，支持不同时间跨度，并用可视化图表来显示。

#	价格	日期
1	2.9069	2023-12-29
2	2.9069	2023-12-28
3	2.9069	2023-12-27

4）查询消息列表



2. 进阶查询需求

1) 查询某商品历史最低价格，支持按不同时间跨度（近一周，近一月，近一年）进行筛选

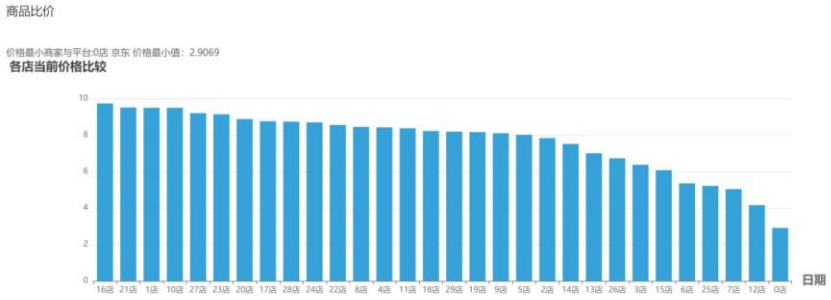
上面已经展示过了，有显示最低价格

#	价格	日期
1	2.9069	2023-12-29
2	2.9069	2023-12-28
3	2.9069	2023-12-27

2) 统计不同用户所收藏商品的分布情况，分析年度热点商品等用户偏好

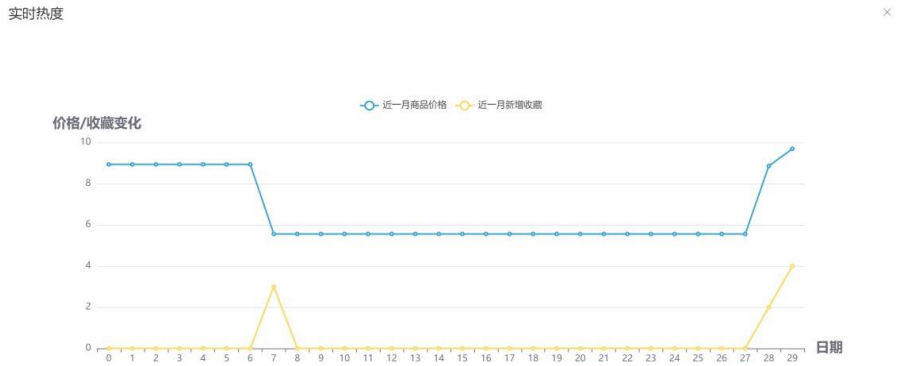
3) 统计每种商品各个商家的价格、在不同时间跨度上的价格差，分析价格差最大最小的商品

4) 统计每种商品在不同商家之间价格差，分析商家间的价格差异情况



5) 其他 1-2 种深度数据分析功能（自行挖掘有趣的需求）

主要是探究实时热度和实时价格的规律，实时热度通过每天的收藏量增加来体现，展示一个月内新增收藏量曲线以及价格变化曲线。但由于测试数据是通过模拟随机数生成的，这里很难看出规律。一般来说，降价时会使得热度高，而热度与收藏量一般是成正比关系，若是正常的大规模用户访问下应该会有这个趋势。



六.系统性能测试分析

1.索引的使用对性能的影响

本数据库对索引的使用主要是在显式地在每个表的主键与外键设置了 BTREE 索引，在同样的网络测试环境下，将前端页面对某大规模数据接口的请求响应时间作为性能测量指标。

注意这里的请求响应时间指广义上的数据打在屏幕上的时间。如对于/product 接口的 GET 方法，对于 concrete_product 表、shop 表、platform 表、product 表有无索引分别进行测试（这里使用的/product 接口是一开始的实现，即还未通过 sql 语句而是 SpringBoot 来实现查找返回，整体速率会比较慢，适合测试）后面发现在建表时设置的对于外键的索引并不允许删除，所以只删掉了主键的索引，但似乎主键本身就带有索引特性，删除后对性能来说也没什么很明显的差别。测试数据如下：

在有主键索引的情况下，/product 接口的 GET 方法进行五次测试，响应时间分别是 5.22s、5.04s、5.24s、5.12s、4.77s，平均时长为 5.08s。无主键索引的情况下，响应时间分别是 4.61s、
3.82s、3.52s、3.45s、3.27s，平均时长为 3.73s。同样的，对于支持搜索的/search 接口的 POST 方法进行测试，有主键索引时五次测试的响应时间分别为 4.19s、3.95s、1.95s、3.66s、1.86s，平均时长为 3.12s。没有主键索引时五次测试的响应时间分别为 2.00s、3.97s、3.90s、4.07s、 4.01s，平均时长为 3.59s。从上述测试来看，显式地为主键设置索引对性能的影响并不明显，甚至对于/product 接口的测试有变快的迹象（当然不排除是网络波动因素），认为应该是主键本身就自带索引特性，所以显式声明并没有什么作用。

2.不同 sql 查询语句对于性能的影响

我们分别比较获取商品简略信息的三种实现方式：后端通过编程语言来筛选数据，sql 查询笛卡尔积来获取关联数据，sql 查询通过连接来获取数据。

第一种：后端通过编程语言来筛选数据，原理是通过 findall 来获取所需的全部数据到内存，然后关联获指定具体商品的店名，商品名，平台名等信息。

```
public List<PartialProductDTO> getAllProductsWithPartialInfo() {
    List<ConcreteProduct> concreteProducts = concreteProductRepository.findAll();
    List<PartialProductDTO> res = new ArrayList<>();
    for (ConcreteProduct product : concreteProducts) {
        PartialProductDTO partialProductDTO = getPartialProductByConcreteProductId(p
        if (partialProductDTO == null) {
            return null;
        }
        res.add(partialProductDTO);
    }
    return res;
}
```

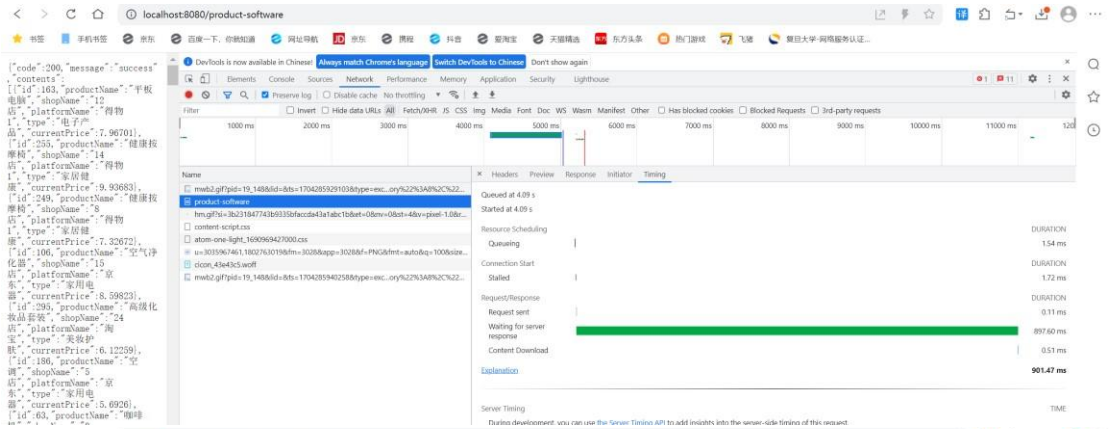
第二种：通过 sql 查询，笛卡尔积来获取所有关联信息：

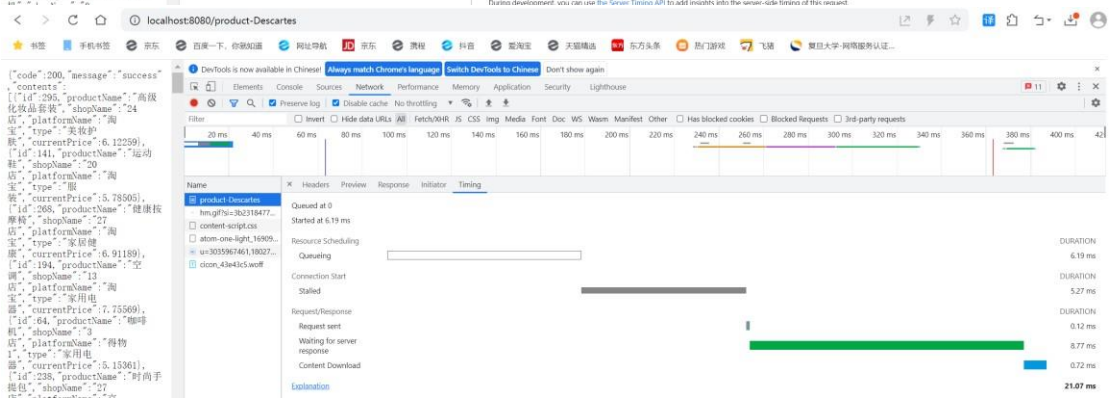
```
@Query("SELECT new com.example.backend0.dto.PartialProductDTO(cp.ID,p.productName,s.shopName,pf.platformName,p.type,cp.currentPrice) " +
       "FROM ConcreteProduct cp,Product p,Shop s,Platform pf " +
       "WHERE cp.productID = p.ID " +
       "and cp.shopID = s.ID " +
       "and cp.platformID = pf.ID "
       )
List<PartialProductDTO> findPartialProductInfoByConcreteIDDescartes();
```

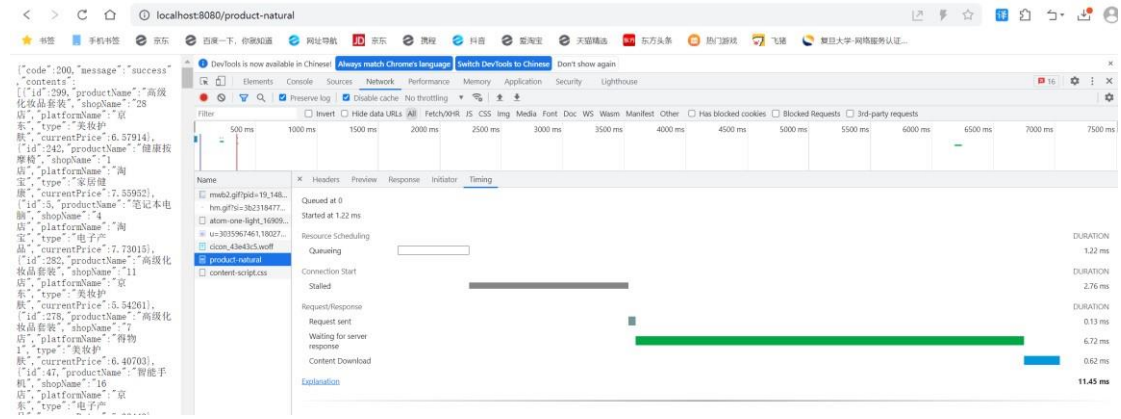
第三种：通过连接来查询关联信息：

```
@Query("SELECT new com.example.backend0.dto.PartialProductDTO(cp.ID,p.productName,s.shopName,pf.platformName,p.type,cp.currentPrice) " +
        "FROM ConcreteProduct cp " +
        "JOIN Product p ON cp.productID = p.ID " +
        "JOIN Shop s ON cp.shopID = s.ID " +
        "JOIN Platform pf ON cp.platformID = pf.ID " )
List<PartialProductDTO> findPartialProductInfoByConcreteIDNatural();
```

分别测试获取数据的时间：







进行重复测量的结果如下：

	1	2	3	4	5	平均
软件方式	904.19 ms	932.29 ms	941.04 ms	885.21 ms	894.84 ms	911.514 ms
笛卡尔积	21.03 ms	14.52 ms	20.34 ms	12.03 ms	18.36 ms	17.248 ms
连接	12.89 ms	14.49 ms	12.13 ms	10.35 ms	11.60 ms	12.328 ms

利用软件的方式，由于一次性读取了所有相关的表的所有数据，有大量多余数据，并且通过代码挨个处理数据，又造成了不必要的时间开销，所以总的传输时长远远大于通过 sql 语句关联查询所需的数据。两种不同的 sql 查询语句所耗费时间开销总体较为接近，可能是 mysql 数据库对于查询语句优化后，到底部操作实现相近，没有真正的使用笛卡尔积联合所有表来查询所需的数据。但总的来看，不同的查询方式对于数据查询的时间影响极大，其中使用连接的 sql 语句查询性能表现最好。